

## **SGD(refresh)**

Loss function:

$$L(y, \hat{y}) = L(x, \hat{y}, \mathbf{w})$$

Predicted output      Input  
Ground truth output      Network weights/parameters (vector)

Gradient:

$$\mathbf{g}(\mathbf{w}) = \frac{\partial L(\mathbf{x}, \mathbf{w})}{\partial \mathbf{w}}$$

Update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{g}(\mathbf{w}_t)$$

Learning rate

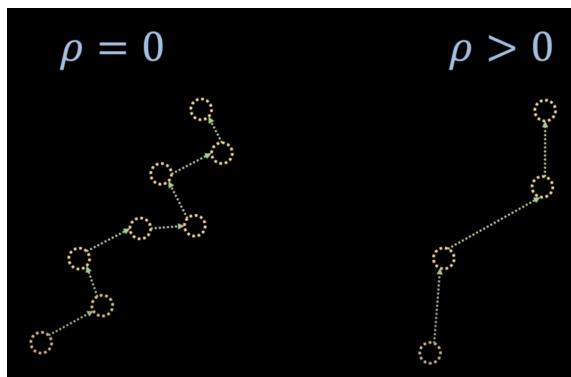
## **MOMENTUM**

Мы использовали:

`tf.keras.optimizers.SGD(momentum=True)`

Idea:

При использовании Momentum, текущий шаг обновления зависит не только от градиента на текущем шаге, но и от накопленного "импульса" предыдущих градиентов. Это похоже на движение периодически подталкиваемого некой силой шара, который на новом шаге продолжает двигаться в направлении вектора силы действовавшей на шар на прошлом шаге по инерции.



Update rule:

$\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{v}_{t+1}$ $\mathbf{v}_{t+1} = \rho \mathbf{v}_t - \eta \mathbf{g}(\mathbf{w}_t)$	$V_{t+1} = \beta V_t + (1 - \beta) \nabla W_t$ $W_{t+1} = W_t - \alpha V_{t+1}$ $\beta = 0.9$
--	---

## ***NESTEROV (Nesterov Accelerated Gradient)***

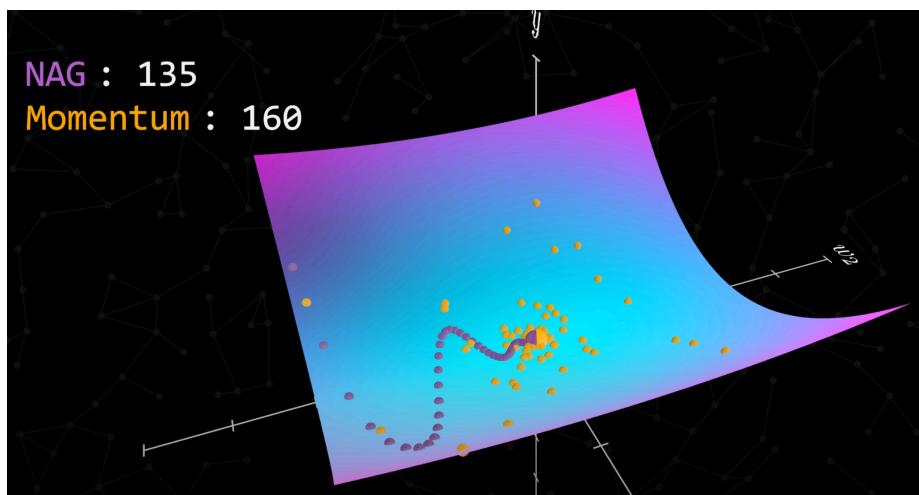
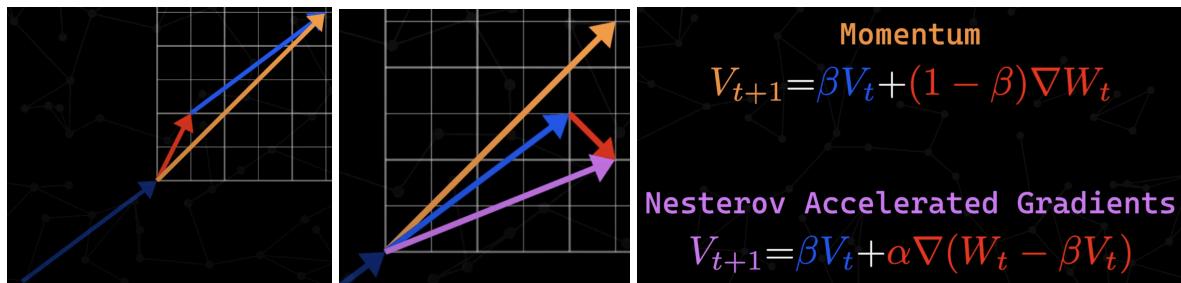
Мы использовали:

`tf.keras.optimizers.SGD(nesterov=True)`

Idea:

Улучшить Momentum, сделать его более точным и стабильным.  
 У Momentum есть идеологическая проблема -> обновление параметров основывается на текущем градиенте, вычисленном в текущей точке. Это может приводить к тому, что накопленный импульс движется в направлении, которое уже "устарело" к моменту следующего шага. Это может приводить к избыточному ускорению в неправильном направлении, особенно когда градиент резко изменяется. NAG частично решает эту проблему, вычисляя градиент в предсказанной точке - в точке после "velocity jump".

Таким образом снижается вероятность чрезмерного ускорения, так как обновление основано на более “актуальной” информации.



Update rule:

$$\begin{aligned}\mathbf{v}_{t+1} &= \rho \mathbf{v}_t - \eta \mathbf{g}(\mathbf{w}_t + \rho \mathbf{v}_t) \\ \mathbf{w}_{t+1} &= \mathbf{w}_t + \mathbf{v}_{t+1}\end{aligned}$$

Params:

**rho** - momentum coefficient

[0.1, 0.25, 0.5, 0.75, 0.9, 0.999]

**betta** - use PyTorch

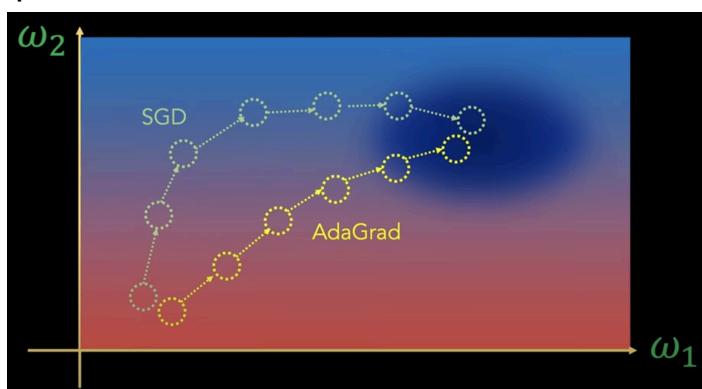
## **ADAGRAD (Adaptive Gradient)**

Мы использовали:

tf.keras.optimizers.Adagrad()

Idea:

В классическом SGD используется фиксированная скорость обучения одинаковая для всех параметров, что может быть неэффективно, особенно если масштабы градиентов разных параметров сильно разнятся. AdaGrad решает эту проблему - покоординатно нормализует градиент, изменяя скорость обучения для каждого параметра индивидуально на основе накопленных градиентов этого параметра. Если у параметра большой суммарный градиент, то нам стоит “притормозить”, если маленький - то “разогнаться”.



Update rule:

$$\mathbf{w} = (w_1, w_2, \dots, w_n)$$

$$\mathbf{v} = (v_1, v_2, \dots, v_n)$$

$$v_{i,t+1} = v_{i,t} + g(\omega_{i,t})^2$$

$$\omega_{i,t+1} = \omega_{i,t} - \frac{\eta}{\epsilon + \sqrt{v_{i,t+1}}} g(\omega_{i,t})$$

Params:

**epsilon** - to prevent division by zero  
[1e-07]

**initial\_accumulator\_value** - value of  $v_{i,0}$   
[0.1, 0.05, 0.25]

## **RMSPROP (Root Mean Square Propagation)**

Мы использовали:

tf.keras.optimizers.RMSprop()

Idea:

В AdaGrad накопленные градиенты непрерывно увеличиваются, что приводит к снижению скорости обучения на поздних стадиях.

RMSProp контролирует размер накопленных градиентов и поддерживает более стабильную скорость обучения на протяжении всего времени путём экспоненциального сглаживания.

Update rule:

$$\begin{aligned} \text{Discount parameter} \\ \mathbf{v}_{t+1} &= \beta \mathbf{v}_t + (1 - \beta) \mathbf{g}(\mathbf{w}_t)^2 \\ \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{\eta}{\epsilon + \sqrt{\mathbf{v}_{t+1}}} \mathbf{g}(\mathbf{w}_t) \end{aligned}$$

Params:

**epsilon** - to prevent division by zero  
[1e-07]

**rho** - discount parameter  
[0.9, 0.75, 0.5, 0.25, 0.1]

**momentum** - momentum coefficient

[0]

**centered** - позволяет учитывать не только второй момент (скользящее экспоненциальное средние квадратов) градиентов, но и первый(скользящее экспоненциальное средние):

$$g_{t+1} = \rho g_t + (1 - \rho) \nabla Q_i(w)$$
$$s_{t+1} = \rho s_t + (1 - \rho) (\nabla Q_i(w))^2$$
$$m_{t+1} = \beta m_t + \frac{\eta}{\sqrt{s_{t+1} - g_{t+1}^2 + \epsilon}} \nabla Q_i(w)$$

$$w = w - m_{t+1}$$

Учет первого момента помогает

корректировать нормализацию второго момента. "May help with training, but is slightly more expensive in terms of computation and memory". Основное применение - зашумленные данные.

[True, False]

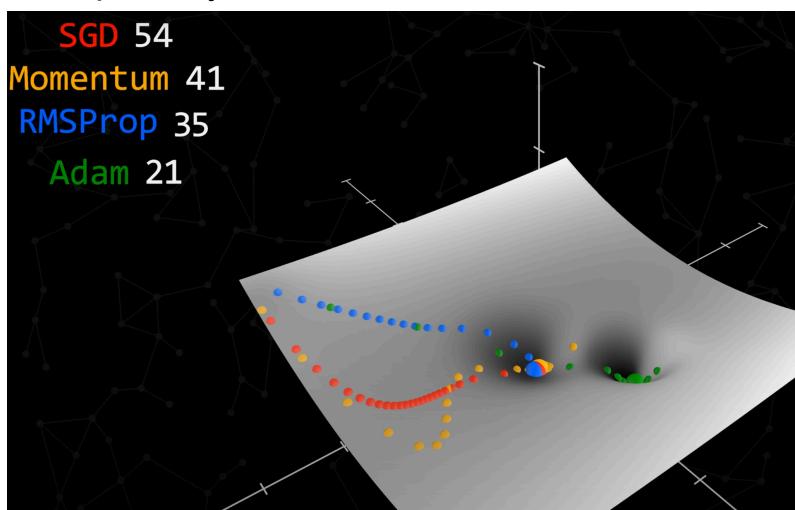
## ***ADAM (Adaptive Moment Estimation)***

Мы использовали:

`tf.keras.optimizers.Adam()`

Idea:

Как следует из названия идея заключается в том, чтобы скрестить две предыдущие идеи.



Update rule:

$$\begin{array}{l|l} \mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \mathbf{g}(\mathbf{w}_t) & \hat{\mathbf{m}}_{t+1} = \mathbf{m}_{t+1} / (1 - \beta_1^{t+1}) \\ \mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2) \mathbf{g}(\mathbf{w}_t)^2 & \hat{\mathbf{v}}_{t+1} = \mathbf{v}_{t+1} / (1 - \beta_2^{t+1}) \\ \mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\epsilon + \sqrt{\hat{\mathbf{v}}_{t+1}}} \hat{\mathbf{m}}_{t+1} \end{array}$$

Params:

**epsilon** - to prevent division by zero

[1e-07]

**beta\_1** -  $\mathbf{m}$  coefficient

[0.9, 0.85, 0.8, 0.75, 0.5]

**beta\_2** -  $\mathbf{v}$  coefficient

[0.999, 0.999999, 0.95, 0.9]

**amsgrad** - на поздних стадиях Adam может демонстрировать ухудшение сходимости из-за очень больших значений вторых моментов. Поэтому полезно будет искусственно ограничивать значение вторых моментов - “AMSGrad uses the maximum of past squared gradients rather than the exponential average to update the parameters”:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

[True, False]

## Common Parameters

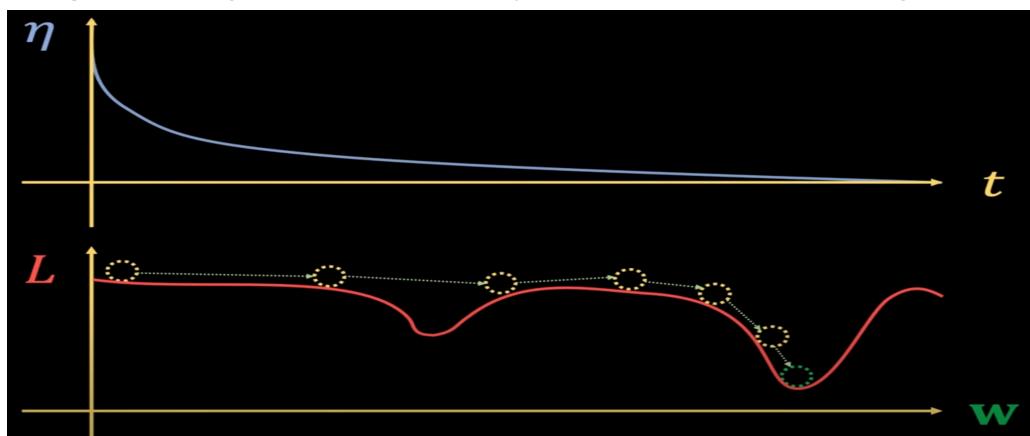
**learning\_rate** - obviously

[0.01, 0.001, 0.0001, 0.00001]

**clipnorm/clipvalue/global\_clipnorm** - to normalize input data

clipnorm = [1]

**weight\_decay** - постепенное уменьшение of learning\_rate



[None]

**loss\_scale\_factor** - special coefficient needed to deal with loss due to finite precision

[None]

**gradient\_accumulation\_steps** - model & optimizer variables will not be updated at every step; instead they will be updated every gradient\_accumulation\_steps steps, using the average value of the gradients since the last update. This is known as "gradient accumulation". This can be useful when your batch size is very small, in order to reduce gradient noise at each update step.

[None]